Solving the Families to Persons Case using EVL+Strace

Leila Samimi-Dehkordi
samimi@eng.ui.ac.ir

Bahman Zamani
zamani@eng.ui.ac.ir

Shekoufeh Kolahdouz-Rahimi
sh.rahimi@eng.ui.ac.ir

MDSE research group
Department of Software Engineering
Faculty of Computer Engineering
University of Isfahan, Iran

Abstract

Benchmarx is the subject of bidirectional transformation case study for the Transformation Tool Contest 2017. The example is a well-known model-to-model transformation from the ATL transformation Zoo named "Families to Persons". This paper presents a solution to provide the inter-model consistency using the Epsilon Validation Language (EVL) and domain-specific traceability techniques. We call this approach EVL+Strace.

1 Introduction

Bidirectional transformations (Bx) are used to restore the consistency when both source and target models are allowed to be modified, but they must remain consistent [1]. A Bx provides two directions from source to target (forward) and from target to source (backward); however, in most approaches, both transformation directions cannot be executed simultaneously [2]. In other words, at each time, only one of the models will be made consistent with the other. Besides that, in most cases, there is more than one way to resolve the inconsistencies. The "Families to Persons" case study [3] is an example of these cases. In this paper, a novel Bx approach called EVL+Strace solves the case study 1. It provides an interactive bidirectional transformation that can execute both directions at the same time and propose more than one fixing ways to restore the consistency. EVL+Strace uses the Epsilon Validation Language (EVL) [4], which expresses constraints between heterogeneous models and evaluates them to resolve the occurred violations. The approach should check if any manual update (deletion, value modification, element relocation, and addition) has occurred in the source or target models. To recognize the type of updates, it is required to store the past information of source and target model in a correspondence (trace) model. This trace model conforms to a metamodel, that we believe it should be specific to the domains of source and target metamodels [5]. EVL+Strace applies EVL on the case-specific trace metamodel to provide a solution for Bx. The rest of the paper is structured as follows. Section 2 describes the EVL+Strace approach. Section 3 presents how the approach solves the case. Section 4 studies the evaluation of the proposed solution. Section 5 concludes the paper.

2 EVL+Strace

The EVL+Strace approach defines EVL modules. Modules consist of a set of invariants (constraints) grouping in the context. An EVL constraint contains two main parts including check and fix blocks. In the check block,
a condition is specified that must be true. If it is evaluated to be false, the fix block triggers some statements to resolve the violation. The Epsilon Object Language (EOL) [6] specifies the checking expressions and fixing statements. The defined constraints in EVL+Strace are applied on the elements of three metamodels including source, case-specific trace, and target. The case-specific trace metamodel defines strongly typed links between source and target meta-elements. The approach code is verbose, and designing a trace metamodel for each transformation case study is time consuming. Therefore, to automatically produce the trace metamodel and generate main parts of code, we implement a tool called MoDEBiTE 2. EVL+Strace can detect independent updates on the source and target models by checking the information of trace model.

Figure 1 illustrates an example of three consistent models. There are two blue stars and a red square in the source model and there exist a blue triangle and a red circle in the consistent target model. Consequently, for each source or target element, there is a link end (TriangleTargetEnd object for the target triangle element) in the trace model. A trace link end keeps the real typed features, the same as features in the corresponding source/target element. There are also a Star2Triangle trace link to relate the star source ends to the triangle target end and a Square2CircleTraceLink link for connecting the square source end with a circle target end.

![Figure 1: Example of consistent models](image)

Figure 2 presents examples of possible atomic updates on source and target models. If the blue star is deleted manually from the source model, the type reference of the star trace link end to the source element will become empty (from the technical view it will convert to an empty EObject). This case of deletion is depicted in Figure 2a. To restore the consistency, it is required to delete the second star from the source model, the triangle from the target model, and the corresponding trace link and link ends from the trace model.

Figure 2b outlines addition of a green circle in the target model. Elements that are not referred by any trace link end are detected as new inserted ones. As depicted, no trace link ends refer to the green star; therefore, EVL+Strace detects it as a new inserted element. To make models consistent, a green square should be inserted in the source model. Moreover, a Square2Circle trace link and corresponding trace link ends with green values must be added to the trace model.

Figure 2c presents the example of value modification. In this case, the color value of the source square is changed from red to yellow. This modification is detected by comparing the value of the corresponding square source end to the color value of the square. When the comparison demonstrates the inequality, the value modification is recognized. To resolve this violation, it is required to update the color value of the target circle and corresponding trace link ends.

Figure 2d presents the element relocation. The reference to the source square is moved from lower star to the upper one. When the relocation is detected, the corresponding reference in the trace model should be changed. The relocation may result in other update operations such as deleting an old element and inserting a new one.

---
2The tool can be downloaded from the MoDEBiTE link in http://mdse.bahmanzamani.com/tools/
3 Solution

In this section, we show how EVL+Strace works using the Families2Persons trace metamodel. Figure 3 shows the case-specific trace metamodel, which is generated automatically by MoDEBiTE. The root of the metamodel is TraceModel. It has two kinds of trace links that are Reg2RegTraceLink and FamilyMember2PersonsTraceLink. The former connects FamilyRegisterSourceEnd to PersonRegisterTargetEnd. The latter links FamilySourceEnd and FamilyMemberSourceEnd to PersonTargetEnd. The instance object of FamilySourceEnd keeps information of the corresponding Family object in the source model such as the value of the name and the references to the MemberSourceEnd objects. To access the corresponding source/target object, the trace link end has a reference type. For instance, FamilySourceEnd defines a familySourceEndType reference, which refers to the Family object in the source model. Note that, the relation between trace links and trace link ends is a bidirectional reference; therefore, accessing the link (end) is possible from the link end (link).

To modularize the EVL+Strace code, some EOL operations are defined for checking or fixing various types of updates. The MoDEBiTE tool automatically generates these operations. The EVL module of EVL+Strace consists of pre block, deletion constraints, modification and relocation constraints, and addition ones. The pre block sets the useXmiIds feature of three resources (models) to true. Through this setting, all created objects in three models have their own xmi:ids. The Bx code deals with objects by means of these ids.

Deletion constraints are defined in the context of the SourceEnds or TargetEnds. They check if a source/target element has been removed and fix the violation by deleting the corresponding TraceLinkEnd instance. In the fix block, the owner link of that link end is notified from this deletion and another constraint in the context of that TraceLink is called. The called constraint deletes all link end objects and their corresponding source/target elements that the TraceLink is referred to. Addition constraints are specified in the context of source/target meta-classes. A new element is an object that has no fingerprint in the trace model. In other words, if there is no trace link end referring to that object, it is detected as a new element. While an addition in source (target) is recognized, the fix block should first add corresponding object in the target (source). Then, it adds corresponding trace link ends for new inserted elements in the trace. At last, it links them by a typed trace link.

Figure 2: Abstract picture of atomic update operations on source and target models
Modification and relocation constraints check and fix modifications of the attribute values. They are specified in the context of the SourceEnds or TargetEnds like deletion constraints. When a modification is recognized by a checking operation, the propagate operation should be called to fix the violation (for instance, modifying the name of Family or Member objects results in changing the name of corresponding Person(s)). Developer may want to delete some objects and add new ones when a modification occurs (modifying the last name of a Person object is an example of this situation). Each reference in source/target model has an equivalent reference in the trace model (if that reference is related to the transformation scenario). For instance, the father, mother, sons, daughters references are defined in the trace metamodel. If a reference of trace refers to a trace link end, and its equivalent reference in the source (target) refers to the object that is not corresponding to the mentioned trace link end, an element relocation is detected. Based on the relocation, a fixing strategy should be defined to restore the consistency.

An example of the EVL+Strace constraint is presented in Listing 1. This constraint checks if the name of the FamilyMemberSourceEnd object is modified (self.nameIsModified()). The nameIsModified() operation compare the names of self object and its correspondence (FamilyMember object in source). If the mentioned values are not equal, then it returns true. When the check expression (negation of the nameIsModified() operation) becomes false, EVL shows the message to the user in the validation view. By right clicking on the appeared message, the title of the fix block is presented to the user. When the user clicks it, the statements of the fix block (here self.namePropagates()) are executed. (to see more examples refer to Appendix A).

```java
context Families2Persons!FamilyMemberSourceEnd{
    guard: not self.isRemoved() and not self.refFamilyMember2Persons.endTypeIsRemoved()
    constraint nameIsModified{
        check: not self.nameIsModified()
        message: 'name of '+self +' is modified'
        fix{
            title:'Propagate the modification'
            do{ self.namePropagates();}
        }}
```

Figure 3: The case-specific trace metamodel for Families to Persons case study
Figure 4 presents an example of three consistent models for the Families to Persons case study. For each Family element (Flanders, Simpson and Skinner) in the source, there exists a corresponding FamilySourceEnd object in the trace model. Likewise, for each member in the source, there is a member in the trace. All source end objects in the trace keep the information (the value of the name attributes) of elements of the source model. Note the relation between FamilySourceEnd elements and FamilyMemberSourceEnd objects, which are the same as the relation between families and members in the source (e.g. the Skinner family has a father Seymour both in the source and trace models). From the target side, for each Person (Male or Female) object, there is an element of the PersonTargetEnd type, which has the same name. A FamilyMember2PersonsTraceLink link connects each family member linkend to a person linkend. Since there is one family register and one person register in the trace model, there exists a single Reg2RegTracelink object to connect them.

3.1 Interactive versus Automatic

EVL+Strace implements an interactive transformation system. In special cases, where there is no conflict between the manual changes on the source and target models, it is possible to specify the constraint in order to be executed automatically. Benefiting from the EVL syntax, Figure 5 shows how to rewrite the constraints to make the auto-fix possible. In automatic EVL+Strace the shape of code is changed, in which fix blocks are removed and their statements are shifted to the check block.

When the number of violations in interactive case is enormous, the user must spend extra effort to select from the alternatives. However, being interactive can be beneficial in check-only mode. In this case, the user may only want to know which constraints are broken, but it is not needed to enforce the consistency. EVL+Strace
does not need to specify the execution mode. The order of selecting the violated messages, which must be fixed, is important in some cases. Therefore, the approach handles execution order by defining some lazy constraints, which is required to be called from other constraints.

4 Evaluation

To test the solution, we use EUnit and Workflow tools [7] of the Epsilon framework. It is required to change the code of EVL+Strace to have automatic behavior. In this case, multiple deletions get the approach into trouble, while the interactive approach can pass this case. To have an automatic transformation, a Configuration metamodel is introduced to preserve the preferExistingToNewFamily and preferParentToChild values. From the Bx tool architecture variability view [8], the proposed approach is an incremental corr-based Bx tool. We use some update examples defined in FamilyHelper.eol and PersonHelper.eol files to provide test cases. Table 1 presents the results of testing EVL+Strace.

<table>
<thead>
<tr>
<th>#</th>
<th>direction</th>
<th>Policy</th>
<th>Change Type</th>
<th>Test Case Name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fwd</td>
<td>fixed</td>
<td>testInitialiseSynchronisation</td>
<td>testCreateFamily</td>
<td>expected pass</td>
</tr>
<tr>
<td>2</td>
<td>fwd</td>
<td>fixed</td>
<td>attribute</td>
<td>testCreateFamily</td>
<td>expected pass</td>
</tr>
<tr>
<td>3</td>
<td>fwd</td>
<td>fixed</td>
<td>add</td>
<td>testCreateFamily</td>
<td>expected pass</td>
</tr>
<tr>
<td>4</td>
<td>fwd</td>
<td>fixed</td>
<td>add</td>
<td>testCreateFamilyMember</td>
<td>expected pass</td>
</tr>
<tr>
<td>5</td>
<td>fwd</td>
<td>fixed</td>
<td>add</td>
<td>testNewFamilyWithMultiMembers</td>
<td>expected pass</td>
</tr>
<tr>
<td>6</td>
<td>fwd</td>
<td>fixed</td>
<td>add</td>
<td>testNewDuplicateFamilyNames</td>
<td>expected pass</td>
</tr>
<tr>
<td>7</td>
<td>fwd</td>
<td>fixed</td>
<td>add</td>
<td>testDuplicateFamilyMemberNames</td>
<td>expected pass</td>
</tr>
<tr>
<td>8</td>
<td>bwd</td>
<td>runtime</td>
<td>add (e ∧ p)</td>
<td>testCreateMalePersonAsSon</td>
<td>expected pass</td>
</tr>
<tr>
<td>9</td>
<td>bwd</td>
<td>runtime</td>
<td>add (e ∧ p)</td>
<td>testCreateMembersInExistingFamilyAsParents</td>
<td>expected pass</td>
</tr>
<tr>
<td>10</td>
<td>bwd</td>
<td>runtime</td>
<td>add (e ∧ ¬p)</td>
<td>testCreateMalePersonAsSon</td>
<td>expected pass</td>
</tr>
<tr>
<td>11</td>
<td>bwd</td>
<td>runtime</td>
<td>add (e ∧ ¬p)</td>
<td>testCreateMembersInExistingFamilyAsParents</td>
<td>expected pass</td>
</tr>
<tr>
<td>12</td>
<td>bwd</td>
<td>runtime</td>
<td>add (e ∧ p)</td>
<td>testCreateDuplicateMembersInExistingFamilyAsChildren</td>
<td>expected pass</td>
</tr>
<tr>
<td>13</td>
<td>bwd</td>
<td>runtime</td>
<td>add (¬e ∧ p)</td>
<td>testCreateMalePersonAsParent</td>
<td>expected pass</td>
</tr>
<tr>
<td>14</td>
<td>bwd</td>
<td>runtime</td>
<td>add (¬e ∧ p)</td>
<td>testCreateMembersInNewFamilyAsParents</td>
<td>expected pass</td>
</tr>
<tr>
<td>15</td>
<td>bwd</td>
<td>runtime</td>
<td>add (¬e ∧ p)</td>
<td>testCreateDuplicateMembersInNewFamilyAsParents</td>
<td>expected pass</td>
</tr>
<tr>
<td>16</td>
<td>bwd</td>
<td>runtime</td>
<td>add (¬e ∧ ¬p)</td>
<td>testCreateMalePersonAsSon</td>
<td>expected pass</td>
</tr>
<tr>
<td>17</td>
<td>bwd</td>
<td>runtime</td>
<td>add (¬e ∧ ¬p)</td>
<td>testCreateMembersInNewFamilyAsChildren</td>
<td>expected pass</td>
</tr>
<tr>
<td>18</td>
<td>bwd</td>
<td>runtime</td>
<td>add (¬e ∧ ¬p)</td>
<td>testCreateDuplicateFamilyMembersInNewFamilyAsChildren</td>
<td>expected pass</td>
</tr>
<tr>
<td>19</td>
<td>fwd</td>
<td>fixed</td>
<td>add</td>
<td>testIncrementalInserts</td>
<td>expected pass</td>
</tr>
<tr>
<td>20</td>
<td>bwd</td>
<td>runtime</td>
<td>del</td>
<td>testIncrementalDeletions</td>
<td>expected pass</td>
</tr>
<tr>
<td>21</td>
<td>fwd</td>
<td>fixed</td>
<td>attribute</td>
<td>testIncrementalRename</td>
<td>expected pass</td>
</tr>
<tr>
<td>22</td>
<td>fwd</td>
<td>fixed</td>
<td>move</td>
<td>testIncrementalMove</td>
<td>expected pass</td>
</tr>
<tr>
<td>23</td>
<td>fwd</td>
<td>fixed</td>
<td>add+del</td>
<td>testIncrementalMixed</td>
<td>expected pass</td>
</tr>
<tr>
<td>24</td>
<td>fwd</td>
<td>fixed</td>
<td>move</td>
<td>testIncrementalMoveRoleChange</td>
<td>expected pass</td>
</tr>
<tr>
<td>25</td>
<td>fwd</td>
<td>fixed</td>
<td>-</td>
<td>testStability</td>
<td>expected pass</td>
</tr>
<tr>
<td>26</td>
<td>fwd</td>
<td>fixed</td>
<td>-</td>
<td>testHippocraticness</td>
<td>expected pass</td>
</tr>
<tr>
<td>27</td>
<td>bwd</td>
<td>runtime</td>
<td>add</td>
<td>testIncrementalInsertsDynamicConfig</td>
<td>expected pass</td>
</tr>
<tr>
<td>28</td>
<td>bwd</td>
<td>runtime</td>
<td>del</td>
<td>testIncrementalDeletions</td>
<td>failure</td>
</tr>
<tr>
<td>29</td>
<td>bwd</td>
<td>runtime</td>
<td>attribute</td>
<td>testIncrementalRenamingDynamic</td>
<td>expected pass</td>
</tr>
<tr>
<td>30</td>
<td>bwd</td>
<td>runtime</td>
<td>del+add</td>
<td>testIncrementalMixedDynamic</td>
<td>failure</td>
</tr>
<tr>
<td>31</td>
<td>bwd</td>
<td>runtime</td>
<td>add</td>
<td>testIncrementalRenamingDynamic</td>
<td>expected pass</td>
</tr>
<tr>
<td>32</td>
<td>bwd</td>
<td>runtime</td>
<td>-</td>
<td>testStability</td>
<td>expected pass</td>
</tr>
<tr>
<td>33</td>
<td>bwd</td>
<td>runtime</td>
<td>-</td>
<td>testHippocraticness</td>
<td>expected pass</td>
</tr>
<tr>
<td>34</td>
<td>bwd</td>
<td>runtime</td>
<td>-</td>
<td>-</td>
<td>expected pass</td>
</tr>
</tbody>
</table>

From all 34 test cases, automatic EVL+Strace approach has 32 expected pass and two failures. The date value in set birthday operations (defined in PersonHelper.eol) is specified by the cal.getTime() statement that returns the date, time (with millisecond), and time zone. The millisecond and time zone are specified based on the current case of the system. Since the expected target models in our test cases are not actively created, the generated and expected target models are only different in two values (millisecond and time zone). In other words, the birthday values of the generated and expected target models are the same in the first parts. Therefore, some results in Table 1 are determined by star(*) which show this case.
5 Conclusion

This paper presents a bidirectional model-to-model transformation solution to the TTC 2017 Families to Persons case study. The proposed solution is based on a novel approach named EVL+Strace, which uses the EVL language (one of the Epsilon family languages) and case-specific trace metamodel. The trace metamodel (correspondence metamodel) is specific to the domains of the Families and Persons case studies. The approach defines constraints to check user updates with the use of EVL. This language enables us to fix the violations if an inconsistency is recognized in constraints. It is possible to program more than one fixing ways, and interactively ask user to restore the consistency. To test the solution, we change the constraints to fix the violations automatically. The evaluation presents that from all 34 test cases, automatic EVL+Strace has 32 expected pass and two failures.

References


A Appendix: Details of our solution

A.1 EOL operations

We divide the EOL operations into four groups including Auxiliary, Delete, Modify, and Add operations. The last three groups have two types, i.e., Check and Fix. Auxiliary operations are used by other EOL operations. They compute values and get or find objects. A Delete operation checks if an object is removed, or in some cases, it deletes an object from models. Following that, Modify operations identifies whether an attribute value is modified, or an element is relocated and then propagates the update. Finally, Add operations investigate if a source/target object is manually added, or in some cases, they insert new elements in models, fix the references, and transform the references from one model into another. For this case study, the MoDEBiTE tool generates 18 auxiliary operations. Three of them compute the values of the name attributes for Family, FamilyMember, and Person objects. Five operations are for getting the source/target objects from the trace link ends and five ones are defined to get the corresponding trace link end for an object in the source/target model. Listing 2 presents examples of auxiliary operations for the case study.

```java
operation computeFamilyName(x:String):String
{ return x.split(', ').first; }
operation computeMemberName(x:String):String
{ return x.split(', ').second; }
operation computePersonName(x:String,y:String):String
```
Additionally, we define eight auxiliary operations (Listing 3) to make programming easier such as `isMale()` operation to check if a member is a male person or `getFamily()` to find the Family object from a member object.

```java
operation Source!FamilyMember isFather():Boolean{ return self.fatherInverse.isDefined();}
operation Source!FamilyMember isMother():Boolean{ return self.motherInverse.isDefined();}
operation Source!FamilyMember isSon():Boolean{ return self.sonsInverse.isDefined();}
operation Source!FamilyMember isDaughter():Boolean{ return self.daughtersInverse.isDefined();}
operation Source!FamilyMember isMale():Boolean{ return self.isFather() or self.isSon();}
operation Source!FamilyMember isFemale():Boolean{ return self.isMother() or self.isDaughter();}
```

Listing 3: Example of user defined Auxiliary operations

The tool generates five operations from the Add-Check category, named `isNew`, to check if a source/target object is new inserted or not. The `isNew` operation for `FamilyMember` object is shown in Listing 4.

```java
@chached
operation Source!FamilyMember isNew(): Boolean
{
    var seq = Families2Persons!FamilyMemberSourceEnd.all.
    select(s|not s.familyMemberSourceEndType.isTypeOf(Families2Persons!EObject));
    for (s in seq)
        if (self.id = s.familyMemberSourceEndType.id)
            return false;
    return true;
}
```

Listing 4: isNew operation for FamilyMember objects

There are 63 operations in the Add-Fix category:
1. Five operations for adding trace link ends (because there are five different types of trace link ends). An example is shown in Listing 5.
2. Two operations for adding trace links (two types of trace links).

```java
operation addFamilyMemberSourceEnd(object: Source!FamilyMember){
    var end = new Families2Persons!FamilyMemberSourceEnd;
    end.familyMemberSourceEndType = object;
    end.name = object.name;
    Families2Persons!FamilyMemberSourceEnd.all.add(end);
    Families2Persons!TraceModel.all.first.linkends.add(end);
    return end;
}
```

Listing 5: addFamilyMemberSourceEnd operation for adding MemberSourceEnd in the trace model

```java
operation addReg2RegTraceLink (srcFamilyRegisterSourceEnd: Families2Persons!FamilyRegisterSourceEnd,
    tarPersonRegisterTargetEnd: Families2Persons!PersonRegisterTargetEnd){
    var link = new Families2Persons!Reg2RegTraceLink;
    link.srcRefFamilyRegister = srcFamilyRegisterSourceEnd;
    link.trgRefPersonRegister = tarPersonRegisterTargetEnd;
    Families2Persons!Reg2RegTraceLink.all.add(link);
    Families2Persons!TraceModel.all.first.links.add(link);
    return link;
}
```

Listing 6: Operations for adding trace links
3. Six operations for inserting new objects in the source and target models. Listing 7 presents the `insertFamilyMember` operation, which takes a `Person` object and creates a `FamilyMember` object in the source.

```java
operation insertFamilyMember (personObject : Target!Person): Source!FamilyMember{
    var source = new Source!FamilyMember;
    source.name = computeMemberName(personObject.name);
    return source;
}
```

Listing 7: insertFamilyMember operation for inserting a member in the source

4. 12 operations for setting source/target references and 12 operations for setting trace references.

5. 24 operations for transforming from source/target references to trace references, and vice versa (Listing 8).

```java
operation Families2Persons!FamilySourceEnd copyFamilySourceEndfather ()
{
    var modelObject = self.getEndType();
    if(self.father.isDefined())
        modelObject.setFather(self.father.getEndType());
}
```

Listing 8: copyFamilySourceEndfather operation transforms elements from trace to source

6. Two operations for transforming from the families reference of FamilyRegisterSourceEnd to the persons reference of PersonRegisterTargetEnd, and vice versa (Listing 9).

```java
operation copyFamilyRegisterSourceEndfamilies2PersonRegisterTargetEndpersons(){
    for(familyRegister in Families2Persons!FamilyRegisterSourceEnd.all)
        for(family in familyRegister.families)
            for(familylink in family.refFamilyMember2Persons)
                familyRegister.refReg2Reg.trgRefPersonRegister.setPersons(familylink.trgRefPerson);
}
operation copyPersonRegisterTargetEndpersons2FamilyRegisterSourceEndfamilies(){
    for(personRegister in Families2Persons!PersonRegisterTargetEnd.all){
        for(person in personRegister.persons)
            personRegister.refReg2Reg.srcRefFamilyRegister.setFamilies(person.refFamilyMember2Persons.srcRefFamily);
    }
}
```

Listing 9: Operations for transforming the relations between SourceEnds and TargetEnds

There are four `Modify-Check` operations for checking if the name attribute is modified or not. One example is presented in Listing 10.

```java
operation Families2Persons!FamilySourceEnd nameIsModified(): Boolean
{
    if(self.name<> self.familySourceEndType.name) return true;
    else return false;
}
```

Listing 10: nameIsModified operation for FamilySourceEnd

For the `Modify-Fix` category, three operations are defined. Listing 11 shows one of these operations.

```java
operation Families2Persons!FamilySourceEnd namePropagates(){
    self.name = self.getEndType().name;
    for (tr in self.refFamilyMember2Persons){
        if(not tr.endTypeIsRemoved())
            tr.trgRefPerson.name = computePersonName(self.name,tr.trgRefPerson.name.split(', ').second);
            var targetObject = Target!Person.all.selectOne(o|o.id = tr.trgRefPerson.personTargetEndType.id);
            targetObject.name = computePersonName(self.name,targetObject.name.split(', ').second);
    }
}
```

Listing 11: namePropagates() operation for FamilySourceEnd

The MoDEBiTE tool generates 12 operations for the `Delete-Check` category including five operations for checking removed source/target objects from the context of trace link ends, five operations for checking trace link ends from the context of trace links and two ones for checking removed trace links. It also produces five operations for deleting the source/target objects and corresponding trace link ends.
A.2 EVL constraints

The pre block sets the xmiId property of resources (Listing 12).

```java
import 'atomicOperations.eol';
pre{
Families2Persons.resource.useXmiIds= true;
Source.resource.useXmiIds= true;
Target.resource.useXmiIds= true;
}
```

Listing 12: pre block of the EVL+Strace code

Deletion constraints check if a source/target object is removed and fix the violation. MoDEBiTE generates 10 constraints for checking and fixing deletions. In Listing 13, the isRemoved constraint is defined in the context of FamilyMemberSourceEnd, and check if a FamilyMember object is removed.

```java
class Families2Persons!FamilyMemberSourceEnd{
constraint isRemoved{
check: not self.isRemoved()
message: 'The '+self+' has a removed type'
fix{
title: 'delete the '+self
var tracelink = self.refFamilyMember2Persons;
delete self;
tracelink.satisfies("srcRefFamilyMemberIsRemoved");
}}
```

Listing 13: isRemoved constraint for FamilyMemberSourceEnd

Modification and relocation constraints check if any attribute value is modified or any element is moved. There are six constraints in this category. Listing 14 demonstrates the code of familyMemberRoleIsRelocated constraint.

```java
class Families2Persons!FamilyMemberSourceEnd{
guard: not self.isRemoved() and not self.refFamilyMember2Persons.endTypeIsRemoved()
constraint familyMemberRoleIsRelocated{
  guard: not self.getEndType().isNew() and
  self.getEndType().getFamily().getTraceLinkEnd()= self.getFamily()
  check: not ((self.fatherInverse.isDefined() and not self.getEndType().fatherInverse.isDefined())
or (self.motherInverse.isDefined() and not self.getEndType().motherInverse.isDefined())
or (self.sonsInverse.isDefined() and not self.getEndType().sonsInverse.isDefined())
or (self.daughtersInverse.isDefined() and not self.getEndType().daughtersInverse.isDefined())
or (self.getEndType().getFamily().getTraceLinkEnd()<> self.getFamily()))
message: self+' role is changed or
'family ='+self.getFamily()+' is changed to '+self.getEndType().getFamily()
fix{
title: 'Propagate the relocation for '+self
do{
  var family= self.getEndType().getFamily();
  var person;
  if((self.fatherInverse.isDefined() or self.sonsInverse.isDefined()) and self.getEndType().isFemale()){
    person =insertFemale(family,self.getEndType());
    person.birthday = self.refFamilyMember2Persons.trgRefPerson.getEndType().birthday;
    delete self.refFamilyMember2Persons.trgRefPerson.getEndType();
    delete self.refFamilyMember2Persons.trgRefPerson;
    self.refFamilyMember2Persons.srcRefFamily = family.getTraceLinkEnd();
    copySrc2Trg();
  }else{
    if((self.motherInverse.isDefined() or self.daughtersInverse.isDefined()) and self.getEndType().isMale()){
      person = insertMale(family,self.getEndType());
      person.birthday = self.refFamilyMember2Persons.trgRefPerson.getEndType().birthday;
      var personTargetEnd = addPersonTargetEnd(person);
      delete self.refFamilyMember2Persons.trgRefPerson.getEndType();
      delete self.refFamilyMember2Persons.trgRefPerson;
      self.refFamilyMember2Persons.srcRefFamily = family.getTraceLinkEnd();
      copySrc2Trg();
    }else{
      person = self.refFamilyMember2Persons.trgRefPerson.getEndType();
      var personTargetEnd = addPersonTargetEnd(person);
      delete self.refFamilyMember2Persons.trgRefPerson.getEndType();
      delete self.refFamilyMember2Persons.trgRefPerson;
      self.refFamilyMember2Persons.srcRefFamily = family.getTraceLinkEnd();
      copySrc2Trg();
    }
  }
}}
```

Listing 14: familyMemberRoleIsRelocated constraint for detecting element relocation
We define 8 constraints for Addition category. Listing 15 represents the code of the \texttt{familyObjectIsNew} constraint. It checks if the family of one member in the source is moved to a new Family object.

<table>
<thead>
<tr>
<th>Context</th>
<th>Source!FamilyMember{}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint</td>
<td>\texttt{familyObjectIsNew} (// the generated code of this constraint should be checked)</td>
</tr>
<tr>
<td>Guard</td>
<td>not self.isNew()</td>
</tr>
<tr>
<td>Check</td>
<td>not ((self.fatherInverse.isDefined() and self.fatherInverse.isNew()) or (self.motherInverse.isDefined() and self.motherInverse.isNew()) or (self.sonsInverse.isDefined() and self.sonsInverse.isNew()) or (self.daughtersInverse.isDefined() and self.daughtersInverse.isNew()))</td>
</tr>
<tr>
<td>Message</td>
<td>self+’ is related to the new family’</td>
</tr>
<tr>
<td>Fix</td>
<td>do{</td>
</tr>
<tr>
<td>Title</td>
<td>'Insert the correspondence’</td>
</tr>
<tr>
<td>Do</td>
<td>var familyMemberSourceEnd = self.getTraceLinkEnd();</td>
</tr>
<tr>
<td>Var family;</td>
<td></td>
</tr>
<tr>
<td>Var familyMember2PersonsLink = self.getTraceLinkEnd().refFamilyMember2Persons;</td>
<td></td>
</tr>
<tr>
<td>Var oldFamilySourceEnd = familyMember2PersonsLink.srcRefFamily;</td>
<td></td>
</tr>
<tr>
<td>Var oldPerson = familyMember2PersonsLink.trgRefPerson.getEndType();</td>
<td></td>
</tr>
<tr>
<td>Var person;</td>
<td></td>
</tr>
<tr>
<td>Family = self.getFamily();</td>
<td></td>
</tr>
<tr>
<td>Family satisfies(&quot;isNew&quot;);</td>
<td></td>
</tr>
<tr>
<td>Var familySourceEnd = family.getTraceLinkEnd();</td>
<td></td>
</tr>
<tr>
<td>Var familyMember2PersonsLink.srcRefFamily = familySourceEnd;</td>
<td></td>
</tr>
<tr>
<td>If</td>
<td>(oldPerson.isTypeOf(Target!Female) and self.isMale()) or (oldPerson.isTypeOf(Target!Male) and self.isFemale())</td>
</tr>
<tr>
<td>Else</td>
<td>person = insertMale(family,self);</td>
</tr>
<tr>
<td>Person.birthday = oldPerson.birthday;</td>
<td></td>
</tr>
<tr>
<td>Delete oldPerson.getTraceLinkEnd();</td>
<td></td>
</tr>
<tr>
<td>Delete oldPerson;</td>
<td></td>
</tr>
<tr>
<td>Var personTargetEnd = addPersonTargetEnd(person);</td>
<td></td>
</tr>
<tr>
<td>FamilyMember2PersonsLink.trgRefPerson = personTargetEnd;</td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td>oldPerson.name = computePersonName(family.name,self.name);</td>
</tr>
<tr>
<td>OldPerson.getTraceLinkEnd().name = oldPerson.name;</td>
<td></td>
</tr>
<tr>
<td>CopySrc2Trg();</td>
<td></td>
</tr>
<tr>
<td>}}</td>
<td></td>
</tr>
</tbody>
</table>

Listing 15: \texttt{familyObjectIsNew} constraint

In automatic EVL+Strace the shape of code is changed, in which fix blocks are removed and their statements are shifted to the check block. Listing 16 shows the excerpt code of this transfiguration. To make the code more readable and modular, we define some new operation and put the statements of fix block in them.

<table>
<thead>
<tr>
<th>Context</th>
<th>Target!Female{}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint</td>
<td>\texttt{isNew} (</td>
</tr>
<tr>
<td>Check</td>
<td>var result = not self.isNew();</td>
</tr>
<tr>
<td>If</td>
<td>(not result){</td>
</tr>
<tr>
<td>If</td>
<td>(preferExistingToNewFamily and Source!Family.all.exists(</td>
</tr>
<tr>
<td>Else</td>
<td>self.fixisNewExistingFamilyParent();</td>
</tr>
<tr>
<td>Else</td>
<td>self.fixisNewExistingFamilyDaughter();</td>
</tr>
<tr>
<td>Else</td>
<td>if(preferParentToChild){self.fixisNewNewFamilyParent();}</td>
</tr>
<tr>
<td>Else</td>
<td>self.fixisNewNewFamilyDaughter();</td>
</tr>
<tr>
<td>Return</td>
<td>true;</td>
</tr>
<tr>
<td>}}</td>
<td></td>
</tr>
</tbody>
</table>

Listing 16: \texttt{isNew} constraint in the context of Female